Logical Operators

Scientific XPL provides three boolean operators:

not	Logical negation
and	Logical and
or	Logical or

The NOT operator requires one operand, while AND and OR require two operands. These operators are most often used when two or more expressions are to be evaluated and then compared in one program statement:

```
if ((i ile ptr) and (y < 0)) then ...
do while ((x < y) or (x > z) or (i = 0));
if ((x > 0) and (not found)) then ...
```

In the first example, the IF expression will evaluate to true if and only if both the expressions combined with AND evaluate to true. In the second example, the expression will be true if one or more of the expressions combined with OR is true. The NOT operator in the third example is used together with a boolean variable to make a comparison. If FOUND is false in the above example, the expression (not found) will evaluate to true; if FOUND is false, the NOT operator negates this, making the final result true.

The XPL compiler will evaluate only as much of a logical expression as is necessary to determine whether the expression is true or false. For example, if any term of an AND expression is false, the entire expression is false. Likewise if any term of an OR expression is true, the entire expression is true. Because the compiler will stop evaluating an expression as soon as the result is known, it is inadvisable to embed a function call that must always occur within a logical expression unless it is the first term. The XPL compiler guarantees that logical expressions will be evaluated left to right.

For example, if the following expression appears in a program:

```
if ((index ile last) and ((not error) or (i <= 5))) then ...
```

and the first term (index ile last one) evaluates to false, evaluation of the whole expression will stop because it is clear that the final result will be false no matter what the other expressions evaluate to. This feature can be used to decrease program execution time if the order of the individual expressions in a complex expression is considered when the expression is written.

Bit Operators

There are seven operators in Scientific XPL that provide bit manipulation capability. These operators are:

not	One's complement
and	Bitwise and
or	Bitwise inclusive or
xor	Bitwise exclusive or
shr	Shift right
shl	Shift left
rot	Rotate left

The NOT operator requires one fixed point operand, while the others require two fixed point operands.

The NOT operator produces a result that is the one's complement of the operand. Each bit of the 16-bit operand is inverted to produce the corresponding bit in the result. For example:

```
not "000000"  =  "177777"
not "177760"  =  "000017"
```

The tilde (~) or the circumflex (^) can also be used to represent the NOT operator.

The operators AND, OR, and XOR compute a 16-bit fixed point result from two fixed point operands. Although the computer performs the function on all 16 bits simultaneously, each bit of the first operand is and'd, or'd, or exclusive or'd with the corresponding bit in the second operand to derive the result.

The ampersand (&) can be used to represent AND, while the vertical bar (¦) or the backslash (\) can be used to represent OR. There is no alternate symbol for XOR.

The bit operators are used in arithmetic expressions in much the same way as their arithmetic counterparts. A single arithmetic expression may include both arithmetic and bitwise operators. For example:

```
i = (i and (not byte_flag));  /* turn off BYTE_FLAG */
j = (j xor 1);                /* toggle LSB of J */
```

The operators SHL, SHR, and ROT also perform bit manipulation on a 16-bit fixed point operand. Calls to these three functions take the following form:

```
i = shl (<value>, <bit_count>);
```

The keyword SHL, SHR, or ROT is followed by two expressions separated by a comma and enclosed in parentheses. Both arguments must evaluate to fixed point results. The returned value is equal to the first argument shifted left (SHL), shifted right (SHR), or rotated left (ROT) BIT_COUNT bit positions. Bits shifted off the left end (SHL) or the right end (SHR) are lost. Bits shifted into the left end (SHR) or shifted into the right end (SHL) will be zeros.

In all cases the bit count must be between zero and fifteen (inclusive) for the result of the shift/rotate to be defined. Do not perform shifts with bit counts less than zero or greater than fifteen.

```
times = shl (times, 1);                          /* multiply by 2 */
i = shl (sects, 8); /* words from sectors (multiply by 256) */
j = shr (words, 8);   /* sectors from words (divide by 256) */
dev = shr (f#ms_sector, 8);   /* get device from upper byte */
byte_swap = rot (value, 8); /* exchange upper & lower bytes */
```

ABLE computers incorporate hardware instructions that perform shift left, shift right, and rotate functions. These instructions operate on a 16-bit quantity, shifting one or eight bit positions per instruction in approximately one microsecond. If the bit count specified in a shift/rotate function is a constant expression, the compiler will compute the appropriate number of shift/rotate instructions and emit them in-line.

The truth tables for AND, OR, and XOR are shown below:

AND

Input Bit A	Input Bit B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

OR

Input Bit A	Input Bit B	A and B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

Input Bit A	Input Bit B	A and B
0	0	0
0	1	1
1	0	1
1	1	0

Examples of AND, OR, and XOR:

```
"000001" and "000000"  =  "000000"
"000003" and "000001"  =  "000001"
"012571" and "177777"  =  "012571"
"012571" and "177776"  =  "012570"

"000001" or "000000"   =  "000001"
"000003" or "000001"   =  "000003"
"012571" or "177777"   =  "177777"
"012571" or "177776"   =  "177777"

"000001" xor "000000"  =  "000001"
"000003" xor "000001"  =  "000002"
"012571" xor "177777"  =  "165206"
"012571" xor "177776"  =  "165207"
```

Assignment Statement

Results of computations are typically stored in variables. At any given moment, a variable has only one value, but this value can change during program execution. The assignment statement respecifies the value of a variable. Its form is:

```
<variable> = <expression>;
```

The expression to the right of the equals sign is evaluated, and the resulting value is assigned to the variable named on the left. The old value of the variable is lost. For example, following the execution of the statement

```
a = 3;
```

the variable A will have a new value of 3.

Numeric Conversions

While evaluating an expression, the XPL compiler is sometimes forced to convert an operand to another data type in order to carry out the operation. There is currently only one implicit conversion that is carried out by the compiler: if either operand of a binary operation is floating point, the other operand will be converted to floating point before the operation (addition, subtraction, etc.) is performed. No information is lost when this conversion is performed.

Likewise if the expression on the right hand side of an assignment statement yields a fixed point result (i.e., the expression involves only fixed point operands), and the receiving variable is of type floating, the fixed point result is converted to floating point before storage. If the result of an expression involving floating point operands is to be assigned to a fixed point variable, the INT function must be used to extract the integer value from the floating result. If the floating result that is passed to the INT function contains a fractional part, information will of course be lost in the conversion from floating to fixed point. Erroneous results will occur using INT if the value of the floating point expression is not in the range of -32,768 to +32,767.

```
dcl (i, j) fixed;
dcl (x, y) floating;

i = i*j;            /* fixed result assigned to fixed */
x = i + j;          /* fixed result assigned to floating */
y = x*i;            /* implicit conversion of I to floating */
i = int (x - y);    /* floating result assigned to fixed - note
                       that the INT function is required */
```

Precedence and Order of Evaluation

Operators in XPL have an implied precedence, which is used to determine the manner in which operators and operands are grouped together. The expression A + B*C causes A to be added to the product of B and C. In this case, B is said to be bound to the operator * rather than the operator +, and as a result, the multiplication will be performed first. In general, operands are bound to the adjacent operator of highest precedence, or to the one on the left in the case of a tie (XPL expressions are evaluated left to right). Valid XPL operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence. Equal precedence operations are evaluated left to right.

```
shr  shl  rot
not
*    /    mod  %    fdiv
+    -
=    ~=   <    <=   >    >=   ieq  ine  ilt  ile  igt  ige
and  or   xor
```

Parentheses should be used to override the assumed precedence. Thus the expression (A + B)*C will cause the sum of A and B to be multiplied by C. For example:

```
a + b + c + d   is equivalent to   ((a + b) + c) + d
a + b * c                            a + (b * c)
a + b - c * d                       (a + b) - (c * d)
a + (b - c) * d                     (a + ((b - c) * d))
```

When using the AND and OR operators in IF and DO WHILE statements, it is imperative that parentheses be used to clarify the desired relationship. For example, the statement:

```
if (a and 2) = 0 then ...
```

directs the computer to AND the variable A with the number 2 and test the result for zero. If it were written without parentheses, the computer would first test the variable A for a true or false condition (by testing A to see if it was even or odd) and then proceed to determine if the number 2 was equal to the number 0 (which of course it is not). Always use parentheses in such situations to avoid errors.

Section IV - Flow of Control

Scientific XPL has several constructs that are used to perform conditional branching, loops, and general control of program execution. By using the constructs described in this section, such as the IF, DO WHILE, and DO CASE statements, the programmer has very specific control over which program statements are executed when, and how many times those statements are executed.

Compound Statements

It is often desirable to group many program statements together and to treat that group as a single statement. This concept is most often used when creating IF or DO WHILE constructs. It is also used to logically group parts of a program together, thus organizing the program and making it easier to read and understand.

The most common way to form a compound statement is to bracket the program statements within the keywords DO and END. Generally the statements within a DO group are indented several spaces (usually three) to improve program readability.

```
do;
   <statement>;
   <statement>;
   <statement>;
end;
```

A bracketed group of statements is regarded as a single statement and may appear anywhere in a program that a single statement may. The DO-END group is most often used to group a number of conditionally executed statements together, such as when it is part of an IF statement.

Sometimes it is desirable to restrict the scope of variables to a small section of the program. In this case, the keyword BEGIN is used rather than the keyword DO.

```
begin;
   <local declarations>;
   <statement>;
   <statement>;
end;
```

Statements grouped together using BEGIN and END are considered a separate program block, in the same way that a procedure is considered a program block. Local variables can be declared inside a BEGIN-END block that have scope only within that block. This concept and how it is used will be discussed further in the section on variable scope.

IF Statement

The IF statement is used to selectively execute one or several program statements based on the result of a conditional expression. The basic form of the IF statement is:

```
if <expression> then <statement>;
```

The expression following the keyword IF is evaluated. If the result is true, then the statement is executed; if the result is false, nothing happens. Control then passes to the statement following the IF construct. Recall that in XPL any odd number evaluates to true, and any even number evaluates to false.

Some examples of the IF statement are:

```
if x < 0 then x = 0;
if (i = 2) and (count >= 10) then i = i + 1;
```

The IF statement can also be used with a compound statement, so that a group of statements are executed if the expression is true. In the next example, the three statements in the DO-END group would only be executed if the expression temp < 100 were true:

```
if temp < 100 then do;
   value = value*2 + 1;
   count = count + 1;
   print 'Count = ', count;
end;
```

If a separate set of program statements should be executed if the condition is false, the ELSE clause should be used. In this case, either the THEN clause is executed (if the condition is true) or the ELSE clause is executed (if the condition is false).

```
if (x < 0) or (x > 100) then do;
   print 'Error - value out of range.';
   x = 0;
end;
else do;
   x = x*2 + 1;
   print 'New value = ', x;
end;
```

The programmer should use caution to avoid dangling ELSE clauses. For example:

```
if i = 1 then if j = 2 then i = i + 1;
else j = j + 1;
```

The ELSE clause above goes with the second IF statement, that is, with IF j = 2, rather than with IF i = 1. To avoid this confusion, the program segment above could be written this way:

```
if i = 1 then do;
   if j = 2
   then i = i + 1;
   else j = j + 1;
end;
```

DO WHILE Loops

The DO WHILE statement is used to specify iterative loops, and takes the following form:

```
do while (<expression>);
   <statement>;
   <statement>;
   <statement>;
end;
```

In this construct, first the expression following the keyword WHILE is evaluated. If the resulting value is true, the statements within the DO-group are executed. When the END statement is reached, the expression is evaluated again, and the sequence of statements is executed again if the value of the expression is still true. The group is executed repeatedly until the expression evaluates to a false value, at which time execution of the statement group is skipped, and program control passes to the statement immediately following the END statement. Remember that in XPL true and false conditions are represented by odd and even values, respectively.

Consider the following example:

```
x = 0;

do while (x <= 4);
   x = x + 1;
end;
```

In the above example, the statement x = x + 1 will be executed exactly five times. When program control passes out of the loop, the value of x will be 5.

A useful form of the DO WHILE statement is as follows:

```
do while (true);
   ...
end;
```

The above example is an infinite loop. Many simple programs need to loop indefinitely performing a specific control function.

An interesting observation should be made at this point. The Scientific XPL compiler produces highly optimized object code, particularly in the area of conditional transfer instructions. Infinite DO WHILE groups such as DO WHILE (TRUE), DO WHILE (1=1), or even DO WHILE (2+2=7-3) produce absolutely no object code. Since the above expressions involve only constants, the compiler can evaluate the true and false condition as the program is being compiled. The XPL compiler will detect such constructs and simplify the object code accordingly.

Iterative DO Loops

An iterative DO loop executes a group of statements a fixed number of times. The simplest form of this construct is:

```
do <variable> = <expression1> to <expression2>;
   <statement>;
   <statement>;
   <statement>;
end;
```

where <variable> is a non-subscripted variable (the loop variable). The effect of this statement is first to store the value of expression1 into the loop variable. If the value is less than or equal to expression2, the DO-END group is executed. The loop variable value is then incremented by one, and the test is repeated. The DO-END group is repeatedly executed until the value of the loop variable is greater than expression2. At this point, the test fails, execution of the DO-END group is skipped, and control passes to the statement after the END statement.

Note that the value of expression2 is evaluated once before the loop is entered and never evaluated again. Futhermore, since an arithmetic signed comparison is used to test for the end of the loop, the difference between expression2 and expression1 cannot be greater than 32,767. Following the execution of a DO loop, the value of the loop variable is undefined.

Some iterative DO loops are similar in effect to DO WHILE loops. For example:

```
do i = 1 to 10;
   x = x + count;
end;
```

The above DO loop is equivalent to the following DO WHILE loop:

```
i = 1;
do while (i <= 10);
   x = x + count;
   i = i + 1;
end;
```

The only difference between these two types of loops is that if expression2 is a variable or other non-constant expression, it is evaluated every time through the DO WHILE loop, but only once (before the loop is entered) for the iterative DO loop.

To increment the loop variable by a value other than one, the optional BY clause can be added to the iterative DO loop:

```
do <variable> = <expression1> to <expression2> by <expression3>;
   <statement>;
   <statement>;
   <statement>;
end;
```

In this case, the loop variable is incremented by the value of expression3, instead of one, each time the END is reached.

```
/* compute the product of the first y odd integers */

dcl (x, y) floating; /* loop variable, limit */
dcl prod   floating; /* product */

y = 4;     /* initialize variables */
prod = 1;

do x = 1 to (2*y - 1) by 2;
  prod = prod*x;
end;
```

Negative loop increments are only allowed if expression3 is a constant. In the case of a negative loop increment, the loop control statement evaluates to true if expression1 is greater than or equal to expression2, as in this example:

```
do i = 100 to 25 by -5;
   total = total + i;
   print i;
end;
```

This loop will be executed for values of i at 100, 95, 90, ..., 30, and 25.

DO CASE Statement

The final form of selective execution is the DO CASE statement. The form of this statement is:

```
do case (<expression>);
   <statement1>;
   <statement2>;
    ...
   <statementN>;
end;
```

The first thing that happens in the above form is the evaluation of the expression following the keyword CASE. The result of this is an integer value K which must lie between 0 and N-1. K is used to select one of the statements of the DO CASE group, which is then executed. The first case (statement1) corresponds to K=0, the second case (statement2) corresponds to K=1, and so forth. After one statement from the group has been selected and executed, control passes beyond the END statement of the DO CASE. If the runtime value of K is greater than the number of cases, then no case is executed and control passes beyond the END immediately. It is good programming practice to explicitly check the limits of the CASE expression before the DO CASE is executed.

Here is an example of the DO CASE statement:

```
do case (score);
   ;                                /* case 0             */
   conversions = conversions + 1;   /* case 1: conversion */
   safeties = safeties + 1;         /* case 2: safety     */
   fieldgoals = fieldgoals + 1;     /* case 3: field goal */
   ;                                /* case 4             */
   ;                                /* case 5             */
   touchdowns = touchdowns + 1;     /* case 6: touchdown  */
end;
```

When execution of this CASE statement begins, the variable SCORE must be in the range of 0 to 6. If SCORE is 0, 4, or 5 then a null statement (consisting of only a semicolon, and having no effect) is executed. Otherwise, the appropriate variable is incremented.

A more complex example of the DO CASE statement is:

```
do case (x - 5);
   x = x - 5;          /* case 0 */
   do;                 /* case 1 */
      x = x + 10;
      y = y + x - 3;
   end;
   ;                   /* case 2 */
   do i = 3 to 10;     /* case 3 */
      a = a + i;
   end;
end;                   /* end of DO CASE */
```

The above example illustrates the use of DO-END blocks to group several statements as a single (although compound) XPL statement.

Statement Labels

Program statements can be labeled for identification and reference. A labeled statement takes the form:

```
<label>: <statement>;
```

where <label> is any valid, otherwise unused, identifier. Here are some examples of labeled statements:

```
LOOP: x = x + 1;
INITIALIZE: i = 0;
```

It is also valid to have a label without a program statement following it, to be used as a target for control transfer. The customary semicolon is not needed in this case:

```
ERROR:
PRINT.MESSAGE:
```

Note the occurrence of the colon (:) after the label in each case. The purpose of a label is to be a documentation and debugging aid, and to provide a target for GOTO statements.

The GOTO Statement

A GOTO statement stops the normally sequential order of program execution by transferring control directly to its target statement. Sequential execution then resumes, beginning with the target statement. The format for the GOTO statement is:

```
goto <label>;
```

where <label> is an identifier which appears as a label in a labeled statement (see above). The effect of the GOTO statement is a transfer of program control directly to the labeled statement. For example:

```
goto ERROR;
```

transfers control to the label ERROR defined above.

A discussion of label scope, which affects the legality of certain GOTO statements, is postponed to the section on scope.

A final note on labels: it is strongly recommended that the IF and DO statements be used instead of labels and GOTO statements wherever possible. The effect in general will be better object code and more readable programs. Only use a GOTO statement if there is no way to organize your program using the IF and DO statements to achieve the desired transfer of control.

Section V - Arrays and Pointers

Scientific XPL provides several data structures for grouping information in manageable ways. One-dimensional arrays allow for creating sets of numbers or characters, as well as the ability to access or change elements individually. Character strings have a special format in XPL, and the BYTE and PBYTE routines allow direct manipulation of string arrays. The DATA declaration defines static sets of numbers (or a string) that can be accessed in much the same way arrays are accessed. Finally, this section will cover pointers, and the direct manipulation of absolute memory locations.

Arrays

It is frequently convenient to let one identifier represent more than one value. Variables that have been declared to represent more than a single data element are called arrays or vectors. Arrays are often used to logically group data elements together, because all data elements can be referred to using the same variable name. XPL currently supports one-dimensional arrays only. A special form of the array structure is also used to store character strings, making character and string manipulation much simpler for the programmer.

The declaration statement for an array must contain the variable name associated with the array, the number of data elements in the array, and the type of the array elements. For example:

```
declare x (100) fixed;
```

This declaration statement causes the identifier X to be associated with 101 data elements, each of type fixed. The 101 data elements may be referred to individually with the names X(0), X(1), X(2), and so on up to X(100). The number in parentheses, which selects the specific data element of the array, is called a subscript or array index. Note that the lower bound of an array is always zero, while the upper bound is the declared size.

It is also possible to declare more than one array of a particular size and type within one declaration statement, such as in the following example:

```
declare (a, b, c) (199) floating;
```

This causes the three identifiers A, B, and C each to be associated with 200 floating point data elements, so that 600 elements of type floating have been declared in all.

The various elements of an array can be accessed and assigned individually by using subscripts as described above. If the third data element is to be added to the fourth, and the result stored in the fifth data element, we can write the assignment statement:

```
x (4) = x (2) + x (3);
```

Remember that the elements of an array are numbered starting with zero, so that X(2) is actually the third element of the array.

Much of the power of an array lies in the fact that its subscript need not be a numeric constant, but can be another variable, or in fact any valid XPL expression. The following program will sum the elements of the array NUMBERS:

```
declare numbers (10) fixed;
declare (sum, i)     fixed;

sum = 0;                        /* initialize variable */

do i = 0 to 10;                 /* loop through all elements */
   sum = sum + numbers (i);     /* add each value to total sum */
end;

print 'The sum of the array = ', sum;
```

Subscripted variables are permitted anywhere XPL permits a simple variable with the one exception that it is not legal to use an array element as the loop variable of an iterative DO group.

The Scientific XPL Compiler does not give an error if the subscript of an array is out of bounds, so care must be taken to check subscript values to make sure they are in range before using them in a program. Accessing array elements that are out of bounds will cause random words of memory to be read or written.

Character Strings

Textual information can be stored in fixed point arrays. A special format of the array structure has been adopted for user convenience and consistency. The zeroeth element of such an array contains the number of bytes used in the array (i.e., the character length of the string). The character data begins at element one of the array, with each word representing two characters (an 8-bit byte is required to store each character). More information on how strings are stored is provided in the appendix "Internal Representations".

Character strings in this format are easily read from and written to the terminal using the LINPUT and PRINT statements. See the sections on these commands for the input and output of character strings.

Scientific XPL offers two built-in functions (BYTE and PBYTE) that assist in the processing of textual information. These functions operate on 8-bit characters stored in a fixed point array in the standard string format.

BYTE and PBYTE

The BYTE function operates on 8-bit data elements stored in a fixed point array. This is most often done when processing characters that were typed in from the terminal (LINPUT) or characters that are destined for the terminal (PRINT STRING). The BYTE function could also be used when data can be expressed in an 8-bit quantity (0-255) and the programmer desires to make more efficient use of memory space.

The BYTE function is used to access one 8-bit byte stored in a fixed point array. The keyword BYTE is followed by the name of the array and the byte number (starting with zero) enclosed in parentheses.

```
declare arr (64) fixed;
declare i        fixed;

linput arr;              /* get character string from terminal */
print byte (arr, 3);     /* print the fourth character */

do i = 0 to 10;          /* print the first eleven characters */
   print byte (arr, i);
end;
```

Note that byte zero is stored in the lower half of element one of the array. ARR(0) always contains the character length of the string.

The PBYTE routine is used to store an 8-bit byte into a specific byte position of a fixed point array. It is used in conjunction with BYTE to process 8-bit numeric quantities. The keyword PBYTE is followed by the name of the array, the byte number (starting with zero), and the new value to store in the specified byte position. Only the appropriate lower or upper byte of the array element is altered.

```
dcl arr (8) fixed;
dcl i       fixed;

do i = 0 to 15;
   call pbyte (arr, i, a.sp); /* fill string w/spaces */
end;

call pbyte (arr, 7, a.p); /* write a P to byte number 7 */
```

The DATA Declaration

Many times it is convenient to reference a list of numeric constants in much the same way one references the elements of an array. XPL provides the DATA declaration so that sets of numbers can be created and then each number can be accessed by position as with array elements. The DATA specifier is used in a declaration statement to create a data list with an initial set of values:

```
declare nlist data (1, 3, 8, 14);
```

The numeric constants enclosed in parentheses are stored into the data list starting with element zero. In the above example, the fixed point numbers will be stored in the four element data list NLIST as follows:

```
nlist (0) = 1
nlist (1) = 3
nlist (2) = 8
nlist (3) = 14
```

The numbers stored in a data list can not be changed during program execution. Each element of the data list is treated as a numeric constant. Assigning new values or changing data list elements is not allowed.

Floating point data lists can also be created by using the attribute FLOATING in a DATA declaration:

```
dcl flist floating data (0, 3, .999, 10.100, .45);
```

A string constant can be assigned a variable name with the DATA declaration:

```
declare version_date data ('1 May 1987');
...
print 'Program version ', string (version_date);
```

The string is stored in XPL string format, with element zero of the array containing the length of the string, and each character being stored as an 8-bit unit. If there is an odd number of characters in the string, then the upper half of the last word in the array will be a zero. For a more detailed discussion of how bytes are packed into a fixed point array, see the appendix "Internal Representations".

Pointers

It is sometimes necessary in complex programming situations to manipulate memory locations directly. XPL provides functions that can be used to locate where variables or arrays are located in memory (ADDR and LOCATION), and also to directly read and write these locations (CORE). The data type POINTER is provided for declaring variables that are to be used as memory pointers.

The ADDR Function

The built-in function ADDR is used to find the absolute memory location of a variable or array element. This makes it possible to read data into the middle of an array or to an absolute location of memory.

The ADDR function is followed by the name of a variable enclosed in parentheses. If the specified variable is an array, then a subscript must be specified to select an array element:

```
dcl ptr pointer;

ptr = addr (count);   /* location of the variable COUNT */
ptr = addr (buf (3));   /* location of element 4 of BUF */
ptr = addr (buf (i + 3));
```

ADDR returns a pointer to the absolute location of memory in which the specified variable is stored.

The CORE Array

Scientific XPL includes a special array called CORE that is used to read and write data from any location of memory. CORE is an array representing internal memory that starts at location zero of memory and has as many elements as there are words of internal memory. CORE is sometimes used in conjunction with ADDR to implement storage allocation routines. Obviously, the user must be careful when writing data into memory to use only locations that are not currently used by the program. More information on program memory can be found in the appendix "Memory Layout".